Baltic J. Modern Computing, Vol. 10 (2022), No. 4, pp. 710–737
https://doi.org/10.22364/bjmc.2022.10.4.07

# Temporal Multi-View Contracts Help Developing Efficient Test Models

Jishu GUIN[1], Jüri VAIN[1], Leonidas TSIOPOULOS[1], Gert VALDEK[2]

[1] Tallinn University of Technology, Ehitaja tee 5, 12616 Tallinn, Estonia
[2] Airobot OÜ, Tallinn, Estonia

Jishu.Guin@taltech.ee, Juri.Vain@taltech.ee,
Leonidas.Tsiopoulos@taltech.ee, gert@airobot.ee

**Abstract.** In this work we focus on practical aspects of test automation, namely reducing the model creation effort for model-based testing by exploiting the multi-view contract paradigm. We take into account explicitly the design view contracts of the system under test and develop test models by views. For integration of view models two options are studied: incremental merging and composition by model conjunction. The view models and their compositions are formalized as Uppaal Timed Automata. The test requirements are expressed as view contracts and formalized in Timed Computation Tree Logic. This allows test model correctness verification against test requirements. As a novel theoretical contribution we extend the notion of assume/guarantee contracts by introducing temporal modalities in contracts. As a second contribution, we demonstrate the feasibility of the approach on an industrial climate control system testing case study. The improvement of testing process productivity is compared to that of developing a monolithic model empirically without extracting views. Finally, we discuss the usability aspects of the method in test development and outline the challenges.

**Keywords:** Model-checking, Model-Based Testing, Contract-based design

## 1 Introduction

Automation in industry has not attained the level of maturity required for a reliable and economically feasible integration testing of cyber-physical systems (CPS). The mainstream methods used in industry for test automation fail to address the complex dynamics caused by the multiple aspects of the system like functionality, timing constraints, security and safety requirements, etc (Törngren and Sellgren, 2018). This demands a method powerful enough to address various design aspects and their integration to produce conclusive result that can assure high standard of end product quality. Current industrial practice in software assurance still mostly relies on manual testing or limited forms of automated testing, e.g., running test scripts as part of continuous integration

process (Dias-Neto et al., 2017; *PractiTest: The 2022 State of Testing Report*, n.d.). The test scenarios are typically designed by test engineers in ad-hoc manner and their quality depends on the creativity of the test designer, his/her intuition, the quality of requirements specifications, etc. The tools used for automated testing often execute randomly chosen scenarios using scripts based on combinations of input data. These test scenarios are limited in scope as they fail to achieve thorough test coverage in terms of parameters like execution paths, range of inputs, timing properties, security properties, etc. These parameters have gained importance especially in complex CPSs because of the dependability requirements of these systems.

Applying model-based automated testing as an alternative to manual testing is a well-known approach to attain the level of rigour that current CPS development demands. The method we have chosen for verification in this work is Model-Based Testing (MBT) which is one of the approaches for automatically generating test cases from a model of the system under test (SUT) (Utting et al., n.d.). However, the main obstacle for the widespread industrial adoption of MBT is the effort required to develop the models. In order to facilitate this process, the methodology demonstrated in this article employs the Contract-based Design (CBD) principle (Benveniste et al., 2015) to modularize the requirements specification and provide more tractable and efficient model driven test suites. Additionally, enhanced performance is attained in the verification of the properties of the system when running verification tools on smaller component models step-wise. In this work we use Uppaal TImed Automata (Uppaal TA) as our modelling language and Timed Computation Tree Logic (TCTL) to specify correctness properties of the models (Behrmann et al., 2004). These properties of test model comprise properties of completeness relative to view contracts and consistency with the specification under test.

As a novel theoretical contribution we extend the notion of assume/guarantee contracts by introducing temporal modalities expressible in TCTL, because temporal and real-time requirements are often the source of misinterpretation and erroneous implementation and need explicit specification in the contracts (Benveniste et al., 2015). As a second contribution, we demonstrate the feasibility of the approach on an industrial climate control system testing case study where two alternative model building approaches, incrementation and conjunction, are applied for composing view models. When applying these approaches the rationale of their usage is discussed from the perspective of application examples. The improvement in productivity of the testing process is corroborated by comparing to that attained by empirically developing a monolithic model in a non-modular manner without clear distinction of views. This resulted in higher complexity of the model due to the loss of clear structure that, in turn, led to reduced efficiency in the modeling process and its verification. Finally, we discuss the usability aspects of the method in test development and outline the challenges.

This article is an extended version of our paper from the proceedings of the 15th International Baltic Conference on Digital Business and Intelligent Systems (DB&IS) (Guin et al., 2022).

The rest of the article is structured as follows. In Section 2 we describe the functionality and requirements of the climate control system considered as the case study. In Section 3 we provide the preliminaries for MBT, for the Uppaal TA formalism and

for CBD. In Section 4 we elaborate on our theoretical extension to temporal multi-view contracts. In Section 5 we present as a reference method the empirical approach to test model development and in Section 6 we present the modular approach to developing test models explicitly elaborating on incrementation and conjunction for composing models. In Section 7 we provide a comparison of the empirical and multi-view contract-based approaches to developing test models, followed by related works in Section 8. In Section 9 we conclude the article.

## 2 Climate control system

The climate control system used to demonstrate multi-view test model construction technique has been chosen intentionally relatively simple though with multiple non-trivial interacting aspects. The system has four components. Two of them implement user interfaces (UI) - Wall Mount Panel and Mobile application. The third unit, Controller, regulates the climate based on the settings provided by the user via the UI and sensor readings. The fourth component, Server, adds cloud access to the system by providing supervisor control services for the controller and mobile application. As shown in Fig. 1 Controller interacts directly with Server and the Wall Mount Panel. The controller hardware consists of sensors and the climate unit to interact with the physical environment. The user is in the loop via the UI components. Server, which is the System Under Test (SUT) in our case study, communicates with multiple controllers distributed over various locations forming a distributed CPS. Server software comprises two main components. One of them serves the Mobile application over HTTP and the other serves the Controllers over a proprietary protocol using TCP sockets. The case study focuses on testing a part of the Server-Controller communication.

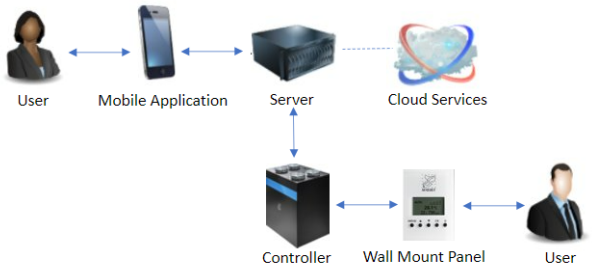


Fig. 1: Components of the climate control system (Guin et al., 2022).

The message interchange between the components has a request-response format. The protocol specification not only covers functional behaviour of the component but also incorporates fault-tolerance, security conditions and time constraints. Each controller can initiate the session independently of others after a successful TCP connection is established with the server. This is followed by exchanging five messages to complete

the connection procedure. The protocol that defines the test scenario includes messages that are exchanged between the server and controller. Each message consists of header and data fields. The values of data fields must correspond to the protocol specification. The sequence of the message exchange for the connection procedure is: 1) Controller initiates the session by sending *client_hello*, 2) Server responds with *server_hello*, 3) Controller sends *init*, 4) Server sends *ack_init*, 5) Controller responds with *ack*.

The commands exchanged after the *hello* message, e.g., *init*, *ack*, etc, are encrypted as a security measure. The encrytion is performed as per the standard Cypher Block Chaining mode which ensures that the messages are sent in multiple equal sized blocks. Transmission of an invalid block to the server leads to disconnection.

In addition to the security measures, Server enforces certain fault-tolerance and timing requirements. After successful execution of the connection procedure the components continue the session by exchanging data messages. The key requirements for the test model development demonstrated in this article are - R1) Server disconnects from the client if it fails to receive *client_hello* within first $I_{TO}$ seconds after successful connection. This is a timing requirement. R2) Server disconnects from the client if the *client_hello* message data fields have incorrect values. This is a fault-tolerance requirement and does not involve any timing constraints. R3) Server shall wait for the first $M_h$ bytes from the client for $I_{TO}$ seconds after the TCP connection is established. The client must be able to send $M_h$ bytes in parts within $I_{TO}$ seconds where $M_h$ is as specified in the system specification. R4) The connection procedure shall successfully complete if all aforementioned requirements are satisfied. The next sections provide an account of the approach used to develop the test models of the described fragment of the SUT.

## 3 Preliminaries

### 3.1 Model-Based Testing

Model-based testing requires a formal model of the SUT composed from system requirements specification. Typically, in MBT the SUT is considered as a black-box which accepts sequences of input data and produces corresponding observable outputs as per the specification (Utting et al., n.d.). MBT provides the test verdict based on the conformance relation between the observable I/O behaviour of the real SUT and its model. The test fails when the behaviours of SUT and the model do not match. The test inputs are generated using assumptions on SUT environment behaviour that are grouped according to pre-specified use cases. The mainstream MBT methods focus on input-output conformance (IOCO) testing. However, since CPS requirements generally refer also to timing constraints, stronger conformance relation that covers also timing, namely RTIOCO (Larsen et al., 2004) relation, is applied in this study.

When choosing the formalism for MBT three main criteria should be met: sufficient expressive power to represent features under test, efficient decidability of the model correctness properties, and relevance for test generation. Model-checking (Baier and Katoen, 2008) is generally used to generate test cases while the coverage criteria are expressed in terms of properties whose validity is verified at first by the model-checker and the witness traces are used thereafter as symbolic test sequences. The test coverage

properties (cf. (Utting et al., n.d.)) can refer to model structural elements such as states and transitions, branching conditions or path expressions.

In general, the coverage properties are extracted from system requirements and expressed in a logic language like TCTL and Linear Temporal Logic. While satisfying the verified property (expressing test coverage criteria in this context), the model-checking witness traces are used thereafter as symbolic test sequences. Executable on real SUT test cases can be extracted from these symbolic traces and instantiated (by means of test adapters) with explicit test data values to reach the test goals. This method is called *offline test generation* because it does not require the SUT to be running at the time when test sequences are generated by model-checking. Converting symbolic traces to executable can also be done *online* by executing the model in random walk mode, but this presumes online compilation of traces to some test scripting language, e.g., TTCN-3, or running the model against SUT via test adapters. The latter, employed also in our approach, requires transforming symbolic test inputs of traces to executable input format of SUT and the outputs of SUT back to symbolic form for conformance check in test execution tool.

### 3.2 Uppaal Timed Automata

The time constraints of the SUT advocate the use of Uppaal TA as the preferred modeling formalism. Uppaal TA (Behrmann et al., 2004) are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into one interacting system by synchronous parallel composition. The nodes of the automata graph are called *locations* and directed arcs between locations are called *edges*. The *state* of an automaton consists of its current location and valuation of all variables, including clocks. Synchronization of processes is defined using constructs called *channels*. A channel *ch* relates a pair of transitions in parallel processes where synchronised edges are labelled with synchronizing input and output actions, e.g., denoted `ch?` and `ch!`, respectively.

Formally, an Uppaal TA is given as the tuple ($L$, $E$, $V$, $CL$, $Init$, $Inv$, $T_L$), where $L$ is a finite set of locations (that denote symbolic states), $E$ is the set of edges (that denote transitions between symbolic states) defined by $E \subseteq L \times G(CL,V) \times Sync \times Act \times L$, where $G(CL,V)$ is the conjunction of enabling constraints, $Sync$ is a set of synchronisation actions over channels and $Act$ is a set of assignments with integer and boolean expressions and clock resets. $V$ denotes the set of variables of boolean and integer type and arrays of those. $CL$ denotes the set of real-valued clocks ($CL \cap V = \emptyset$). $Init \subseteq Act$ is a set of initializing assignments to variables and clocks. $Inv : L \rightarrow I(CL,V)$ maps locations to the set of invariants over clocks $CL$ and variables $V$. $T_L : L \rightarrow \{ordinary, urgent, committed\}$ maps locations to location types. In *urgent* locations the transition denoted by the outgoing edge will be executed immediately when its guard holds. *Committed* locations are useful for creating sequences of actions executed atomically without time passing.

#### 3.2.1 Uppaal TA Requirement Specification Language.

The requirement specification language (in short, query language) of Uppaal TA, used to specify properties to

be model-checked, is a subset of TCTL (Behrmann et al., 2004). The query language consists of path formulae and state formulae. State formulae describe individual states, whereas path formulae quantify over execution paths of the model and can be classified into *reachability*, *safety* and *liveness* (Behrmann et al., 2004). For example, safety properties are specified with formula $\mathsf{A}\,\Box\,\varphi$ stating that first order state formula $\varphi$ should be true in all reachable states that is expressed with the pair of modalities $\mathsf{A}\,\Box$.

For real-time applications, *time bounded reachability* is one of the most fundamental properties. In Uppaal TA, the reachability of a state which satisfies formula $\varphi$ from model initial state is expressible using TCTL formula pattern $\mathsf{A}\,\Diamond\varphi$ && $Clock \leqslant TB$, for time bound $TB$. A special case of time bounded reachability is the reachability of a state when it is considered relative to some other preceding state of the model. This is expressed in TCTL using the "leads to" operator as $ts \rightsquigarrow_{TB} rs$, for preceding state $ts$ and reachable from this state $rs$ within time bound $TB$. In this article we use time bounded reachability to introduce temporality to formulas of assume/guarantee contracts.

### 3.3 Contract-based Design

The CBD paradigm has proven essential for the development of complex systems with many parallel and heterogeneous components adhering to various safety and timing constraints in addition to their basic functionality (Benveniste et al., 2015). Contracts handle components´ interface properties representing the *assumptions* on their environment and the *guarantees* (regarding output) of the component under these assumptions. The main advantage of CBD is the explicit identification of responsibilities of the individual components within a complex system. This facilitates component reuse and scalability while addressing correctness and system complexity through components´ and their services´ operations. The complexity of CPSs requires separation of design concerns by introducing *multi-view* contracts to support compositional design, testing and verification.

The *meta-theory* of contracts introduced by Benveniste et al. (Benveniste et al., 2015) defines interface contracts as abstraction subject to contract algebra. A contract $C$ can be defined in terms of an environment in which it operates and of a component that implements it. A contract is said to be *consistent* if there is a component implementing it and *compatible* if there is an environment in which the contract can operate.

For the case study in this paper we are concerned with two main *contract operators* complementing each other. The first one, the *composition* operator between two contracts, denoted by $\otimes$, is a partial function on contracts involving a *compatibility* criterion. Two contracts $C$ and $C'$ are *compatible* if their shared variable types match and if there exists an environment in which the two contracts properly interact. Working with *Assume/Guarantee* (A/G) contracts being pairs $(A,G)$, the composition $C_1 \otimes C_2 = (A_{C_1\otimes C_2}, G_{C_1\otimes C_2})$ of two contracts $C_1 = (A_{C_1}, G_{C_1})$ and $C_2 = (A_{C_2}, G_{C_2})$ is defined as follows:

$$G_{C_1\otimes C_2} = G_{C_1} \wedge G_{C_2},\ A_{C_1\otimes C_2} = \max\left\{A \,\middle|\, \begin{matrix} A \wedge G_{C_1} \Rightarrow A_{C_2} \\ A \wedge G_{C_2} \Rightarrow A_{C_1} \end{matrix}\right. \quad (1)$$

where "max" refers to the order of predicates by implication and $A_{C_1 \otimes C_2}$ is the weakest assumption such that the two referred implications hold. The two contracts $C_1$ and $C_2$ are compatible if the assumption computed as above differs from *false*.

The second contract operator required when merging different view contracts is the *conjunction* operator. Conjunction (denoted $\wedge$) of contracts is complementary to composition. Full specification of a component can be a conjunction of multiple viewpoints, each covering a specific aspect (behavioural, timing, safety, etc.) of the intended design and specified by an individual contract. Similarly to design modularization by view contracts, the same can be exploited for test model development and its modularization that will be demonstrated in the rest of the paper.

## 4 Temporal multi-view contracts

For the work in this paper we consider the contracts to be in *saturated form* (Benveniste et al., 2015) where the assumptions imply the guarantee.

$$A_{sat} = A,\ G_{sat} = A \Rightarrow G \quad (2)$$

However the contracts of form (2) express static view of functionality without distinguishing different states of the component in which the assumption or guarantee must hold. To make these temporal aspects explicit we extend the above to a temporal saturated contract where:

$$A_{sat} = A,\ G_{sat} = A \rightsquigarrow G \quad (3)$$

When the explicit timing aspects need to be expressed in the contracts then the "leads to" in (3) is strengthened to *time-bounded leads to* ($\rightsquigarrow_{TB}$) and we get formula (4) where both $A$ and $G$ can be expressed in TCTL.

$$A_{tsat} = A,\ G_{tsat} = A \rightsquigarrow_{TB} G \quad (4)$$

For contract *composition* with saturated contracts, the guarantees of each component are explicitly the assumptions of the other:

$$G_{C_1 \otimes C_2} = G_{C_1} \wedge G_{C_2},\ A_{C_1 \otimes C_2} = (G_{C_1} \Rightarrow A_{C_2}) \wedge (G_{C_2} \Rightarrow A_{C_1}) \quad (5)$$

In our approach, since we need the contracts to express also temporal and timing properties, we extend the saturated contract composition formula (5) to *temporal saturated contract formula* (6) by substituting the implication "$\Rightarrow$" with leads to "$\rightsquigarrow$".

$$G_{C_1 \otimes C_2} = G_{C_1} \wedge G_{C_2},\ A_{C_1 \otimes C_2} = (G_{C_1} \rightsquigarrow A_{C_2}) \wedge (G_{C_2} \rightsquigarrow A_{C_1}) \quad (6)$$

However, this extension does not come for free. Since the distribution law of $\wedge$ and $\rightsquigarrow$ does not hold in temporal contract, formula (6) is not linearly extendable. Nevertheless, extension of formula (5) with "leads to" allows compositional verification by checking the individual conjuncts of (6) independently.

Regarding contract *conjunction* with saturated contracts, assume a contract $C \equiv \wedge_{i=1,n} C_i$ comprising views each of them characterised with its own subcontract $C_i \equiv A_{C_i} \rightsquigarrow G_{C_i}$. Proving conjunction $\wedge_{i=1,n} C_i$ correctness means then proving each conjunct $C_i$ separately, at first, and thereafter proving non-interfence of individual view contracts by proving their pairwise conjuncts. Though this increases the number of verification tasks the complexity of each is kept lower (and more likely feasible for the model-checker) than proving full $C$ at once. The operators´ applications are exemplified in Section 6.

## 5 Empirical approach to model-based test generation

To demonstrate the advantages of multi-view contracts-based test model development compared with empirical approach we present, at first, empirical in this section and multi-view approach in Section 6. Fig. 2 shows the Uppaal TA model of the server, the SUT of our case study described in Section 2. The server model captures only the session initialization procedure where different design views manifest themselves clearly. In addition to the I/O behaviour of SUT, the model also introduces some additional channels that do not influence SUT I/O behaviour but are needed for implementing the test execution and communication with test adapters.

The empirically constructed SUT model represents behaviours that cover requirements R1-R4 identified in Section 2. Location *Cnctd* in Fig. 2 is reached when the server has established TCP connection with the controller. The outgoing edges from location *Cnctd* model the four scenarios being subject to requirements. The transition to location *DC* occurs when the controller fails to send the *client_hello* message within $I_{TO}$ seconds after socket connection is established. Location *BH* is reached if the *client_hello* message is received with incorrect values in its fields. The transition to location *PH* is taken in case of partial reception of *client_hello* message. Finally, the transition to location *Hello* are taken when the *client_hello* message is received successfully by the server. Test case generation step introduces the four corresponding environment models each to test one of these scenarios.

The four test scenarios based on the requirements R1-R4 are implemented at first empirically as separate SUT environment models (cf. Fig 2.c - Fig 2.f). These models are selected to specify the test cases of SUT (Fig 2.a) execution based on the actions of the controller model as shown in Fig. 2.b. Auxiliary variable *hs* with value range [1,4] is used to select the test case to be executed. The edge connecting location *Cnctd* to *W0* triggers the *go* signal. The activated test case model identified by the value of *hs* initiates the test on reception of *go* signal. On test completion the controller disconnects from the server and the test case selection cycle repeats with same or different scenario. Thus, for running integration tests that capture alternative behaviours in random order in each cycle a scenario is selected non-deterministically for execution.

Fig. 2.e shows the model for the scenario where the controller fails to send the *client_hello* message within the specified time window. The model execution waits for $I_{TO}$ time units in location *W0* after the test is initiated by *go* signal. The model as shown in Fig. 2.d executes the test scenario where the controller sends an invalid *client_hello* message. The variable *bf* is assigned a random value between [0x01,0x0F]. The bits

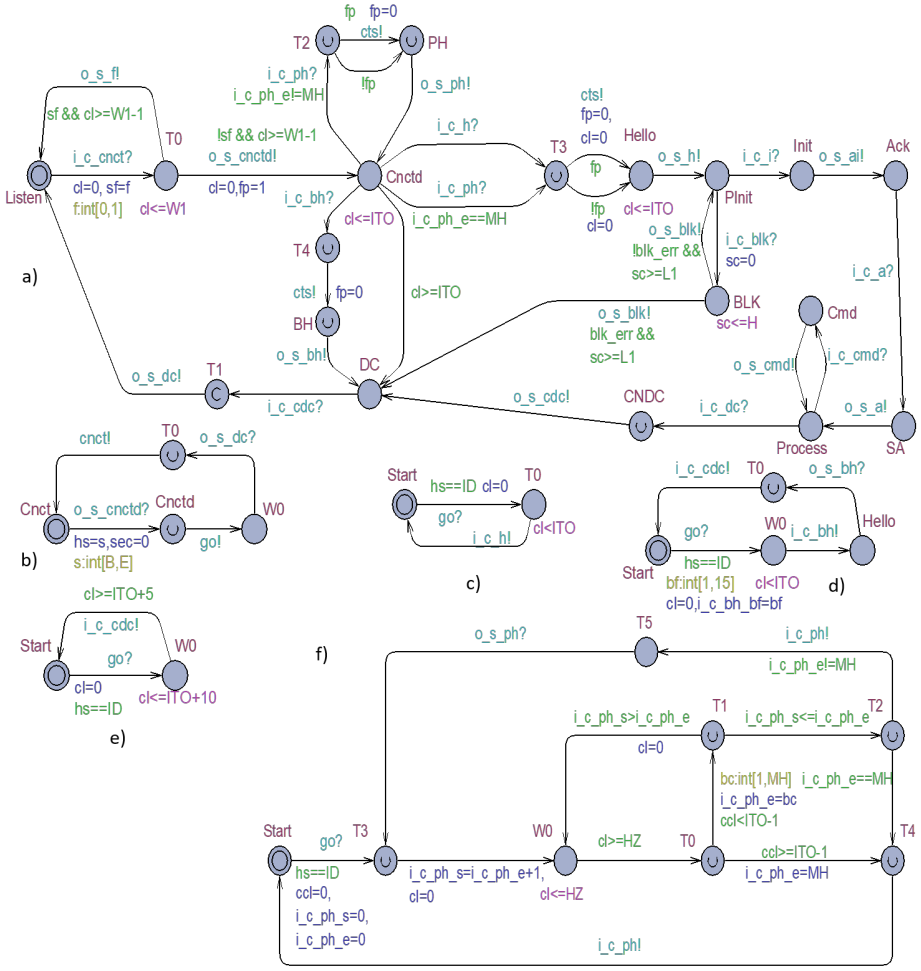


Fig. 2: Empirical model for behaviour, fault-tolerance and timing.

in the lower nibble are mapped to four fields of the message. A value of 1 in any field would generate an invalid *client_hello* with an incorrect value for that field. Thus, the test can generate messages with different number of invalid fields.

The third test scenario is shown in Fig. 2.f. The scenario represents the transmission of *client_hello* message by the controller to the server in parts till all $MH$ ($= M_h$) bytes are sent to the server. The model executes in a loop and randomly chooses the parts of the data field till all bytes are transmitted. The model also ensures that all bytes must be transmitted within the time window ITO(=$[0, I_{TO}]$).

Finally, the model as shown in Fig. 3 depicts the scenario of a correct transmission of the *client_hello* message from the client ensuring that *i_c_h* is triggered before the time window ITO expires. *client_hello* message is sent to the server when the signal is triggered. In this branch of testing the security aspect of the system is modelled. Subsequent commands after the controller has received *server_hello* messages are transmitted as encrypted blocks. The controller model triggers *i_c_blk* from location *PInit* to send *n* encrypted blocks sequentially to the server. The transmission stops if a block encryption error occur. *blk_err* denotes the block encryption error condition. An error triggers the disconnection flow in the server model as per requirement.The full model developed for the work includes few other processes that play supporting role in the test scenario, e.g., to retry connection to server based on timing requirements for the task.

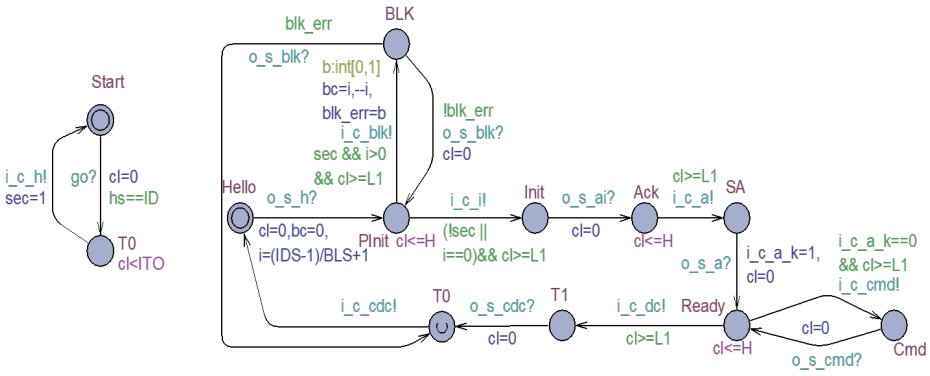


Fig. 3: Empirical model for security.

## 6 Test model modularization by multi-view design contracts

### 6.1 Operations of composing view models

This section elaborates modularization principles that allow one to split the large monolithic test models into smaller and conceptually more homogeneous parts. Namely, we apply the multi-view approach by formalizing requirements as view contracts and build the test models by each view individually. This allows compositional verification to

assure that each test model representing a view satisfies the view contract individually that entails the correctness of the entire conjunction. However, since the feature interaction of individual view models cannot be always guaranteed, due to potential non-orthogonality of some views, the non-interference of their conjunctions needs to be assured by checking their correctness properties conjoining them pair-wise. But due to the view contracts partiality their implementation models (and their pairs) have generally smaller size compared with holistic non-modular models.

For better clarification of the modelling process applied in the case study we represent the multi-view model development scenarios as sublattices of the lattice that reflects possible sequences of multi-view model construction operation applications (cf. Fig. 4). Dotted arrows in the figure represent operations that are possible but not used in the case study and solid lines represent the operations actually applied in the case study test model development. Two operations are used, view models *incrementing* and *conjunction*. Both transformations will be clarified as follows.

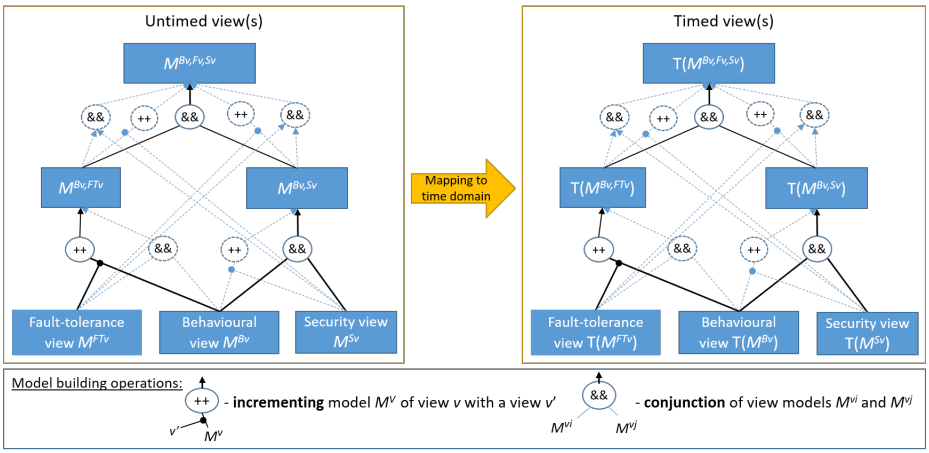


Fig. 4: Multi-view model building scenario options.

Let a set $V = \{Bv, FTv, Sv\}$ denote a set of system untimed views where $Bv$, $FTv$, $Sv$ stand for the behavioural, fault-tolerance and security views we consider in this article, respectively, and let $\mathrm{T}(V) = \{\mathrm{T}(Bv), \mathrm{T}(FTv), \mathrm{T}(Sv)\}$ denote the timed views of $V$.

Having defined the set $V$ (respectively $\mathrm{T}(V)$), $M^v$ denotes a model of view $v \in V$, ($M^{Tv}$ denotes a model of timed view $Tv \in \mathrm{T}(V)$). Then model incrementing operation $M^v + + v'$ means supplementing a model $M^v$ of view $v$, with elements that model the view $v' \in V \cup \mathrm{T}(V)$, where $v' \neq v$ and neither $v$ or $v + + v'$ include an untimed view $v$ and its timed counterpart $Tv$ at the same time.

Alternatively, the conjunction $M^{vi}$ && $M^{vj}$ of view models $M^{vi}$ and $M^{vj}$ presumes that both view models exist before applying the conjunction. Since $M^{vi}$ and $M^{vj}$ may have same submodel $M^{vx}$ of some view $vx$ involved in earlier operations (possibly there may be even more than one of such submodels), then duplicates will be removed in

the conjunction $M^{vi}$ && $M^{vj}$. Recall that conjunction does not guarantee correct result by default, therefore non-interference test should be applied after each conjunction operation to show that the original properties of both conjuncts are still preserved in the conjunction. It is also important to note that both $++$ and && operators are distributive w.r.t. introducing timing conditions to the model, i.e., $\mathrm{T}(M^{v}) \; ++ \; \mathrm{T}^{v'} = \mathrm{T}(M^{v++v'})$ and $\mathrm{T}(M^{v})$ && $\mathrm{T}(M^{v'}) = \mathrm{T}(M^{v\&\&v'})$. So mapping of view models to the timing domain (incrementing with timing respectively) can be applied on view models both before or after they are composed.

### 6.2 From multi-view contracts to view model composition operations

Based on the modular modelling process we introduce design view oriented contracts as an intermediate step to guide the modularization of requirements specification and construction of test models. The derivation of contracts is done in two steps, where firstly, a set of requirements is derived from the textual specification and, secondly, a set of view contracts are formulated based on those requirements. This process is manual and does not guarantee a strict bijective relation between the requirements and contracts however it provides a traceable relation between their groups. To further facilitate the presentation of the multi-view modelling approach we proceed by first presenting the incremental multi-view model development before we elaborate on the development of the orthogonal design view´s conjunction for this case study and show how it is composed with the incremental views. The views exemplified in the rest of the paper are the *behavioural*, *timing* and *fault-tolerance* views, while the *security* view is orthogonal to the other views. Note that any view or composition of untimed views can be incremented to timed view without restrictions. In this case study we have incremented all other views with *timing* view for exposing explicitly the system timing requirements. It is also important to distinguish the timing constraints that are extracted from design requirements from those that are needed to assure the model correctness according to Uppaal TA semantics, e.g., for ensuring time progress in the model. In the following, the timing constraints of requirements are explicitly stated in the contracts and transfered to timed view models.

### 6.3 Behavioural view

The behavioural view highlights the functional requirements extracted from the system specification. In order to encode the requirements in the form of contracts, we define the symbols that are used to denote certain concepts as described in Table 1.

Tables 3 and 4 show the behavioural view contracts for Controller and Server, respectively. Contract $B_1^c$ in Table 3 asserts that if Controller, identified by $id$, is not connected to Server, $\neg\, A_{con}(id)$, where $A_{con}(id)$ represents the state of the connection with controller($id$) in the list of connections $A_{con}$, then it moves to a state attempting to connect to Server, $S_{cl}(id) = connecting$. $B_1^c$ corresponds to requirement $R_1^b$ in Table 2, stating "When controller is not connected, it tries to connect". Contract $B_1^s$ in Table 4 denotes the counter part of this transaction on Server side. It states that given the Controller is connecting to Server, $S_{cl}(id) = connecting$, a transition is made by Server from a connectionless state, $\neg\, A_{con}(id)$, to a connected state, $A_{con}(id) \wedge S_{con}(id) = connected$, where

Table 1: Symbols

| Symbol | Description |
|---|---|
| $id$ | Identifier of a controller |
| $S_{cl}(id)$ | State of the controller with identifier $id$ |
| $I_{TO}$ | Connection timeout for the *hello* message |
| $M_h$ | Number of bytes in *hello* message |
| $S_{con}(id)$ | State of the dedicated connection on the server side to the controller($id$) |
| $data\_error(id)$ | Value denoting occurence of error in the data exchanged with controller($id$) |
| $A_{con}$ | Set of active connections between server and controllers |
| $I_f$ | Time alloted for fast connection retries |
| $I_s$ | Short periodicity for faster connection attempts |
| $I_l$ | Long periodicity for connection attempts |
| $i$ | Number of the current block of encrypted data |
| $n$ | Total number of blocks in a command |
| $S_{sc}(id)$ | State of the controller security module with identifier $id$ |
| $S_{ss}(id)$ | State of the server side security module for the connection with controller($id$) |
| $block\_error(i)$ | Value denoting occurence of error in the encryption of the block number $i$ |

$S_{con}(id)$ represents the state of the dedicated connection from server to controller($id$). The other behavioural contracts formulated in the tables capture the remaining functional requirements as listed in Table 2, regarding exchange of the *hello* message and *init* command. In both Tables 3 and 4 the mapping to the different behavioural requirements from Table 2 is indicated in the first column. The timing and fault-tolerance aspects of the specification are not considered in this view. However, the contracts reflect the causality aspect of the system which is expressed by the temporal operator "leads to" ($\rightsquigarrow$). In behavioural view contracts ideal lossless communication is assumed. Therefore, send and receive predicates of the same message are counted logically identical in this view.

Table 2: Behavioural requirements

| Ref. | Description |
|---|---|
| $R_1^b$ | When controller is not connected, it tries to connect |
| $R_2^b$ | When controller is connected, it sends clear text *client_hello* message |
| $R_3^b$ | After receiving a valid *client_hello*, server responds with *server_hello* message |
| $R_4^b$ | After receiving *server_hello* from server client will send encrypted *INIT* command |
| $R_5^b$ | After receiving *INIT* command, server sends encrypted *ACK_INIT* command |
| $R_6^b$ | After receiving *ACK_INIT* client sends *ACK* |

Table 3: Controller behavioural view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $R_1^b$ | $B_1^c$ | $\neg A_{con}(id)$ | $\neg A_{con}(id) \rightsquigarrow S_{cl}(id) = connecting$ |
| $R_2^b$ | $B_2^c$ | $S_{con}(id) = connected$ | $S_{cl}(id) = connected \rightsquigarrow S_{cl}(id) = chello\_sent$ |
| $R_4^b$ | $B_3^c$ | $S_{con}(id) = shello\_sent$ | $S_{cl}(id) = shello\_recv \rightsquigarrow S_{cl}(id) = init\_sent$ |
| $R_6^b$ | $B_4^c$ | $S_{con}(id) = ackinit\_sent$ | $S_{cl}(id) = ackinit\_recv \rightsquigarrow S_{cl}(id) = ack\_sent$ |

Table 4: Server behavioural view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $(R_1^b)$ | $B_1^s$ | $S_{cl}(id) = connecting$ | $\neg A_{con}(id) \rightsquigarrow A_{con}(id) \wedge S_{con}(id) = connected$ |
| $R_3^b$ | $B_2^s$ | $S_{cl}(id) = chello\_sent$ | $S_{con}(id) = chello\_recv \rightsquigarrow S_{con}(id) = shello\_sent$ |
| $R_5^b$ | $B_3^s$ | $S_{cl}(id) = init\_sent$ | $S_{con}(id) = init\_recv \rightsquigarrow S_{con}(id) = ackinit\_sent$ |
| $(R_6^b)$ | $B_4^s$ | $S_{cl}(id) = ack\_sent$ | $S_{con}(id) = ack\_recv \rightsquigarrow S_{con}(id) = cmd\_ready$ |

**6.3.1 Behavioural view model.** The behavioural view model is presented in Fig 5. The controller automaton is on the left of the figure while the server automaton is in the middle. On the right, the automaton that models the connection protocol is depicted. In this view, the server just responds positively to the controller connection requests, i.e., connection loss and data corruption are abstracted away from this view. The states of the corresponding view contracts are encoded as locations in the automata and communication actions between the components are represented with channels. State names follow a shorthanded notation in the model. For example, controller state "$S_{cl}(id) = chello\_sent$" maps to location name "CHSent" in the controller model.

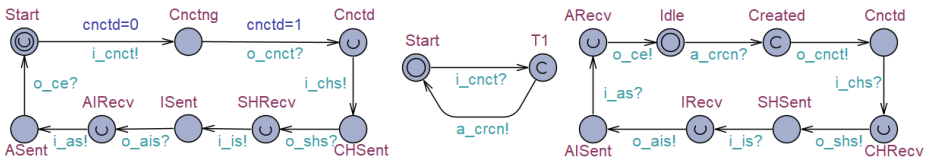


Fig. 5: The behavioural view model of the climate control system (Guin et al., 2022).

**6.3.2 Behavioural view model verification.** In order to verify the model against the specified contracts, we map the contracts to TCTL queries and apply the UPPAAL model-checker. This process is exemplified next with a corresponding pair of contracts

for the controller and the server. Contract $B_2^c$ is mapped to TCTL query:

$$(Con.Cnctd \text{ and } C.Cnctd) \dashrightarrow C.CHSent \quad (7)$$

stating that when the connection is established, this leads to the state where the controller has sent the *client_hello* message. Contract $B_2^s$ is mapped to query:

$$(C.CHSent \text{ and } Con.CHRecv) \dashrightarrow Con.SHSent \quad (8)$$

stating that the server will send the *hello* message after receiving the *hello* message from the controller. Both these queries are satisfied, as well as all other contract queries for this view. Notice the saturated temporal contract *composition* where the guarantee of contract $B_2^c$ is the assumption of contract $B_2^s$:

$$G_{B_2^c \otimes B_2^s} = ...G_{B_2^c} \wedge G_{B_2^s}..., A_{B_2^c \otimes B_2^s} = ...(G_{B_2^c} \rightsquigarrow A_{B_2^s}) \wedge (G_{B_2^s} \rightsquigarrow A_{B_3^c})... \quad (9)$$

Thus, contract composition compatibility and consistency verification requires model-checking the satisfiability of $G_{B_2^c \otimes B_2^s}$ and $A_{B_2^c \otimes B_2^s}$ by the model where both SUT and environment corresponding view models are composed. Recall that contract assumption strengthens the left hand side of the *leads to*. This does not violate the contract semantics under given modelling assumption - lossless communication.

### 6.4 Fault-tolerance view

The climate control system requirements also address scenarios of faulty communication between Controller and Server. The connection procedure specification as described in Section 2 refers to faults related to network link and data fields. Like in behavioural view, the contracts for fault-tolerance view are extracted from requirements. The main difference is that server component's environment is set of controllers that communicate with server via non-ideal media, i.e., the effect of lossy links and data corruption should be explicitly reflected in the assumptions of the server contracts. The fault-tolerance requirements are shown in Table 5. Requirement $R_1^f$ specifies that Controller must retry to connect in the event of disconnection. The event may occur due to network failure or transmission of invalid data. $R_1^f$ in effect is the same as requirement $R_1^b$ from the point of view of Controller, i.e., when the controller is disconnected it tries to connect. Since contract $B_1^c$ maps to requirement $R_1^b$ for Controller and therefore $R_1^f$, the fault-tolerance view of the component remains unchanged. $R_2^f$ is relevant for Server and augments Server's behavioural view with information pertinent to fault-tolerance.

Server's fault-tolerance view contracts are shown in Table 6. Contracts $F_1^s$ and $F_2^s$ concern the succesful and unsuccesful connection of Server and Controller, respectively, depending if there is connection error or not. Contracts $F_3^s$ and $F_4^s$ concern the succesful and unsuccesful sending and receiving of the *hello* message, respectively, depending if there is data error or not.

Table 5: Fault-tolerance requirements

| Ref. | Description |
|---|---|
| $R_1^f$ | Controller retries to connect if disconnected |
| $R_2^f$ | Server disconnects if data received is invalid |

Table 6: Server fault-tolerance view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $(R_1^b)$ | $F_1^s$ | $S_{cl}(id) = connecting$ $\wedge \neg conn\_error(\mathrm{id})$ | $\neg A_{con}(id) \rightsquigarrow A_{con}(id)$ $\wedge S_{con}(id) = connected$ |
| $(R_1^f)$ | $F_2^s$ | $S_{cl}(id) = connecting$ $\wedge conn\_error(id)$ | $\neg A_{con}(id)$ |
| $R_3^b$ | $F_3^s$ | $S_{cl}(id) = chello\_sent$ $\wedge \neg data\_error(id)$ | $S_{con}(id) = chello\_recv \rightsquigarrow$ $S_{con}(id) = shello\_sent$ |
| $R_2^f$ | $F_4^s$ | $S_{cl}(id) = chello\_sent$ $\wedge data\_error(id)$ | $S_{con}(id) = chello\_recv \rightsquigarrow$ $\neg A_{con}(id)$ |

**6.4.1 Fault-tolerance view model.** The fault-tolerance view model is shown in Fig 6. In addition to the behavioural view model there is an environment component (middle up) which affects, first, how the server (middle low) responds to the controller in case of connection error and, secondly, how the later stages of the connection proceed regarding data error. The controller (left) and connection (right) models are enriched with transitions to handle these fault cases.

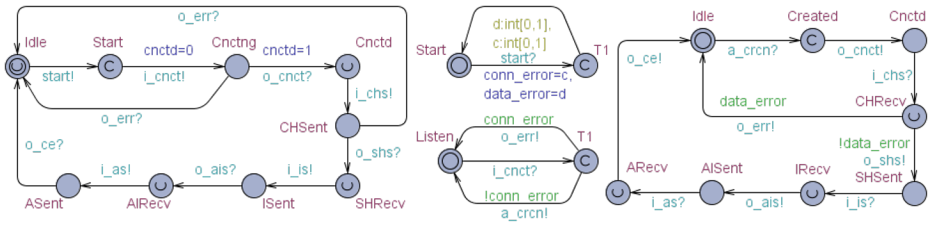


Fig. 6: The fault-tolerance view model of the climate control system (Guin et al., 2022).

**6.4.2 Fault-tolerance view model verification.** Alike the behavioural view, the fault-tolerance view contract verification is exemplified next with one pair of corresponding

contracts for the interacting components. Contract $F_4^s$ is mapped to TCTL query:

$$(C.CHSent \text{ and } data\_error \text{ and } Con.CHRecv) \dashrightarrow !cnctd \quad (10)$$

stating that when the controller has sent the *hello* message and the server has received it with data error, this leads to the server disconnecting. Contract $B_1^c$ is mapped to query:

$$!cnctd \dashrightarrow C.Cnctng \quad (11)$$

stating that when the controller is not connected, it tries to connect.

### 6.5 Timing view

The timing view of the climate control system addresses scenarios influenced by timing requirements. Timing requirements concern frequency at which the controller retries connections with the server, resetting the clock, and waiting times. The timing view requirements are shown in Table 7. $R_1^t$ specifies that the controller must retry to connect every $I_s$ minutes for the first $I_f$ minutes after disconnection or start up and $R_2^t$ complements the requirement by stating that connection is retried every $I_l$ minutes thereafter. Requirement $R_3^t$ is about the timer reset after the first data transfer and requirement $R_4^t$ specifies the maximum allowed waiting time for Server to receive the full *hello* message. The requirements are mapped to Controller contracts $T_1^c$ to $T_4^c$ in Table 8 and Server contracts $T_1^s$ to $T_4^s$ in Table 9. For example, contract $T_4^c$ states that when the connection is established, this leads to Controller sending the *hello* message in less than $I_{TO}$ seconds.

Table 7: Timing requirements

| Ref. | Description |
|---|---|
| $R_1^t$ | The connection is retried every $I_s$ minutes for $I_f$ minutes |
| $R_2^t$ | The connection is retried every $I_l$ minutes after $I_f$ minutes |
| $R_3^t$ | The timer is reset on first data transfer over a successful connection |
| $R_4^t$ | Server waits for $I_{TO}$ seconds to receive $M_h$ bytes of *client_hello* message |

**6.5.1 Timing view model.** The timing view model is shown in Fig 7. Since timing is not orthogonal with other views it can be merged with both behavioural and fault-tolerance views separately. Since fault-tolerance view is already built upon the behavioural view we superimpose the timing straight on the fault-tolerance view. The timing additions can be seen on the controller (left) and connection (right) model. The rest of the model remains the same. The connection might time out before the controller

Table 8: Controller Timing view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $R_1^t$,$R_2^t$ | $T_1^c$ | $conn\_error(id)$ | $S_{cl}(id)$= $connecting \rightsquigarrow S_{cl}(id)$= $wait$ |
| $R_1^t$ | $T_2^c$ | $C_g \leq I_f$ | $S_{cl}(id)$= $wait \rightsquigarrow_{=I_s} S_{cl}(id)$= $connecting$ |
| $R_2^t$ | $T_3^c$ | $C_g > I_f$ | $S_{cl}(id)$= $wait \rightsquigarrow_{=I_l} S_{cl}(id)$= $connecting$ |
| $R_2^b$,$(R_4^t)$ | $T_4^c$ | $S_{con}(id)$= $connected$ | $S_{cl}(id)$= $connected \rightsquigarrow_{<I_{TO}} S_{cl}(id)$= $chello\_sent$ |

Table 9: Server Timing view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $(R_1^b)$ | $T_1^s$ | $S_{cl}(id)$= $connecting \wedge \neg conn\_error(id)$ | $\neg A_{con}(id) \rightsquigarrow A_{con}(id)$ $\wedge C_h = 0 \wedge S_{con}(id)$= $connected$ |
| $R_4^t$ | $T_2^s$ | $S_{cl}(id)$= $connected \wedge C_h \geq I_{TO}$ | $\neg A_{con}(id)$ |
| $R_3^b$ | $T_3^s$ | $S_{cl}(id)$= $chello\_sent$ $\wedge \neg data\_error(id) \wedge C_h < I_{TO}$ | $S_{con}(id)$= $chello\_recv$ $\rightsquigarrow S_{con}(id)$= $shello\_sent$ |
| $R_2^f$,$R_3^t$ | $T_4^s$ | $S_{cl}(id)$= $chello\_sent$ $\wedge\ data\_error(id)$ $\wedge C_h < I_{TO}$ | $S_{con}(id)$= $chello\_recv$ $\rightsquigarrow(\neg A_{con}(id) \wedge C_g = 0)$ |

has sent the *client_hello* message. This is modelled with the upper transitions of both the controller and the connection automaton. In case connection error occurs initially, there is new dedicated part in the controller model involving locations "Wait", "W1" and "W2" handling explicitly the timing requirements for the connection retrials. The rest of the timing requirements are about the time bounded completion of the connection protocol modelled with the clock invariant and clock guards in the lower transitions of the controller and connection automata.

**6.5.2 Timing view model verification.** The timing view contract verification is exemplified next with one pair of related contracts for the interacting components. Controller contract $T_4^c$ is mapped to query:

$$(Con.Cnctd \text{ and } C.Cnctd \text{ and } ch == 0) \dashrightarrow (C.CHSent \text{ and } ch < I_{TO}) \quad (12)$$

stating that when the connection has been established this leads to the controller sending the *hello* message in less than $I_{TO}$ seconds. Server contract $T_3^s$ maps to query:

$$(C.CHSent \text{ and } !data\_error \text{ and } ch < I_{TO} \text{ and } Con.CHRecv) \dashrightarrow Con.SHSent \quad (13)$$

stating that when the controller has sent the *hello* message without data error in less than $I_{TO}$ seconds, this leads to the *hello* message sent by the server. All contracts for this view are satisfied according to the UPPAAL verifier report.

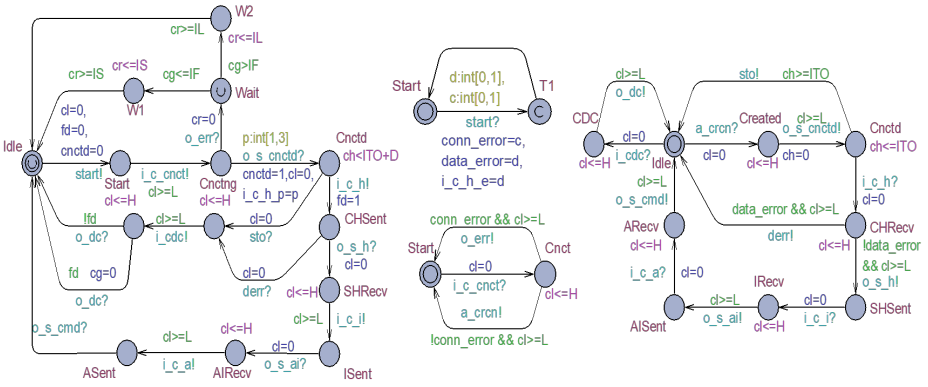


Fig. 7: The timing view model of the climate control system (Guin et al., 2022).

### 6.6 Consistency of incremental multi-view contracts

While in general the multi-view contracts consistency has to be verified taking the conjunction of view contracts (recall that due to scalability constraints in this approach it is done pairwise), in the incremental approach it suffices verifying only the contract of the last increment that is the strongest, and that entails all the contracts of all preceding increments. Let us show this with an instance of a view contract for the server that was specified for the behavioural view, then was refined to the fault-tolerance view and finally supplemented with timing constraints. Behavioural view contract $B_2^s$ is conjoined with fault-tolerance view contract $F_3^s$ and with timing view contract $T_3^s$:

$$\begin{aligned} & C.CHSent \ \wedge \ Con.CHRecv \rightsquigarrow Con.SHSent \\ \wedge \ & C.CHSent \ \wedge \ Con.CHRecv \ \wedge \ \neg data\_error \rightsquigarrow \ Con.SHSent \\ \wedge \ & C.CHSent \ \wedge \ Con.CHRecv \ \wedge \neg data\_error \ \wedge \ ch < I_{TO} \rightsquigarrow \ Con.SHSent \end{aligned} \tag{14}$$

The consistency of the rest of the incremental multi-view contracts was verified in the same manner.

### 6.7 Security view

The messages following the *hello* messages are encrypted in the system. These encrypted messages are sent as equal sized blocks as required by the standard Cypher Block Chaining mode of encryption. The size of the block $N$ is defined by the specification and number of blocks $n$ is determined by the size of the message.

In this work the security view is demonstrated on the *init* messages which follows the *hello* message. Security view model is created as a separate model orthogonal to rest of the system and then composed with the timing view model which is the final increment including the fault-tolerance increment of the bahavioural view. The requirements for the security view are shown in Table 10. Requirement $R_1^s$ states that commands are

Table 10: Security requirements

| Ref. | Description |
|---|---|
| $R_1^s$ | Commands are sent and received in encrypted blocks of size $N$ |
| $R_2^s$ | Block encryption errors lead to immediate disconnection |

sent and received in encrypted blocks of size $N$ and requirement $R_2^s$ states that an error in the block creation and encryption leads to disconnection.

The Controller and Server view contracts are shown in Tables 11 and 12, respectively. Contracts $S_1^c$ to $S_3^c$ for Controller and contracts $S_1^s$ to $S_3^s$ address requirement $R_1^s$ handling the first, intermediate and final block sending. Contract $S_4^s$ addresses requirement $R_2^s$ for immediate disconnection if block error occurs.

Table 11: Controller security view contracts

| R.Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $R_1^s$ | $S_1^c$ | $S_{ss}(id)$= *Ready* | $S_{sc}(id)$= *Ready* $\rightsquigarrow$ $S_{sc}(id)$= *Sent*(1) |
| $R_1^s$ | $S_2^c$ | $S_{ss}(id)$= *Recv*($i$) | $i < n \rightsquigarrow S_{sc}(id)$= *Sent*($i$+1) |
| $R_1^s$ | $S_3^c$ | $S_{ss}(id)$= *Ready* | $S_{sc}(id)$= *Sent*($n$) $\rightsquigarrow$ $S_{sc}(id)$= *Ready* |

Table 12: Server security view contracts

| R. Id | C. Id | Assume | Guarantee |
|---|---|---|---|
| $R_1^s$ | $S_1^s$ | $S_{sc}(id)$= *Sent*($i$) $\wedge$ $\neg$*block_error*($i$) $\wedge$ $i < n \wedge i > 1$ | $S_{ss}(id)$= *Recv*($i$-1) $\rightsquigarrow$ $S_{ss}(id)$= *Recv*($i$) |
| $R_1^s$ | $S_2^s$ | $S_{sc}(id)$= *Sent*(1) $\wedge$ $\neg$ *block_error*(1) | $S_{ss}(id)$= *Ready* $\rightsquigarrow$ $S_{ss}(id)$= *Recv*(1) |
| $R_1^s$ | $S_3^s$ | $S_{sc}(id)$= *Sent*($n$) $\wedge$ $\neg$*block_error*($n$) | $S_{ss}(id)$= *Recv*($n$-1) $\rightsquigarrow$ $S_{ss}(id)$= *Ready* |
| $R_2^s$ | $S_4^s$ | $S_{sc}(id)$= *Sent*($i$) $\wedge$ *block_error*($i$) | $\neg A_{con}(id)$ |

**6.7.1 Security view model.** The security view model is created as a separate model because it is orthogonal to the rest of the system and it is shown in Fig 8. The process starts when Controller (middle) receives the command from the environment (left) on channel *cmd*. Then Controller starts sending one by one $n$ encrypted blocks to Server (right) and each time Server acknowledges the receiving of blocks. If there is block encryption error the process is terminated (shown in lower parts of Controller and Server automatons) and the connection is closed in turn.

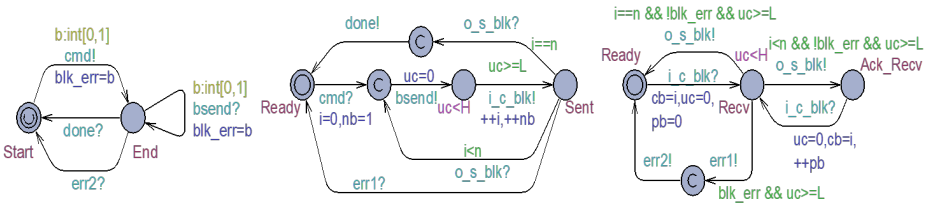


Fig. 8: The security view model of the climate control system.

**6.7.2 Security view model verification.** Let us exemplify the security view contract verification with a pair of corrsponding contracts for Controller and Server. Controller contract $S_1^C$ maps to query:

$$S.Ready \text{ and } C.Ready \dashrightarrow (C.Sent \text{ and } i == 1) \quad (15)$$

stating that when both Controller and Server are ready to start the process, this leads to Controller sending the first block. Server contract $S_2^S$ maps to query:

$$C.Sent \text{ and } i == 1 \text{ and } !blk\_err \dashrightarrow S.Recv \text{ and } S.cb == 1 \quad (16)$$

stating that when Controller has sent the first block and there is no block error, this leads to Server receiving it (Server's current block variable *cb* is equal to 1).

### 6.8 Composition of incremental and orthogonal views

In this work the orthogonal security view model is created as a separate model to the rest of the system views which can then be composed with the other system models constructed incrementally in different configurations depending on testing goal and scenarios.

For this case study the security view model is applied on the *init* message which follows the *hello* message. The composed model of the security view with all the integrated incremental views developed previously is shown in Fig 9. In the upper part of the figure the Controller and the Server timing view model is shown and in the lower part of the figure the security view model is shown.

In order to compose the models, relevant updates are required to sunchronise them. These updates are seen in the lower right part of each timing view automaton and in the environment automaton of the security view in the lower left part of the figure. Spliting of existing transitions and addition of sunchronisation channels *scmd* (for security command), *sdone* (for security done) is applied. Also addition of transitions with synchronisation channels *serr* (for security error), *dc* (for disconnect) is needed to close the connection if encryption block error occurs. The rest of the models, i.e., their original functionality is unchanged.

**6.8.1 Verification of the composition consistency.** In order to verify the consistency of the composed orthogonal and incremental view models, all the contracts from each

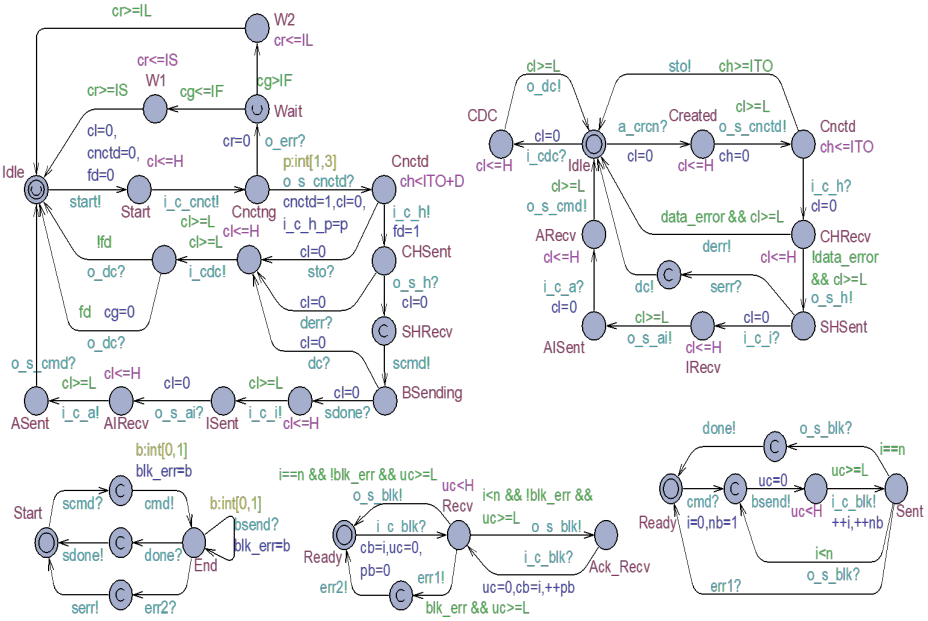


Fig. 9: The composed timing view and security view model of the system.

view composed need to be checked by the UPPAAL verifier. If each of these contracts is satisfied on composition model it means that their conjunction is satisfied as well and there is no interference. For the composed timing and security view model all except two of the individual view contracts are satisfied. The updates to the unsatisfied view contract queries are as follows.

Query for contract $S_1^C$ as exemplified in the security view above needs to be updated to query:

$$S.Ready \text{ and } C.Ready \dashrightarrow (C.Sent \text{ and } i == 1) \text{ or } conn\_error \text{ or } data\_error \text{ or } blk\_err \tag{17}$$

in order to consider also the options that can interfere the correct sending of encrypted blocks from Controller to Server.

Query for contract $B_3^C$ is updated to query:

$$(Con.SHSent \text{ and } C.SHRecv) \dashrightarrow C.ISent \text{ or } blk\_err \tag{18}$$

because the *init* message from Controller to Server cannot be sent if there is block encryption error. This is reflected in the second disjunct of the *leads to* right hand side.

## 7 Comparison of empirical and multi-view contract-based testing

In this section, the usability aspects of the test model development method proposed in Section 6 are compared with those of empirical approach (Section 5). Here we discuss

how the criteria used for comparison are defined/understood, at first, and, second the characteristics of used criteria are evaluated based on the experience gathered by practical application of both test model development methods on the climate control system (cf. Section 2). The following criteria are used for comparison: *test model construction effort*, *test model correctness verification effort*, *test deployment effort*, *test execution effort*, and *root cause analysis effort*. The comparison results presented in this section are also in line with the identified benefits of the CBD in general when coping with complex systems (Benveniste et al., 2015).

The empirical model and the views models formalize the same set of requirements. In the empirical method the requirements are taken as is from the text specification and modelled without intermediate steps and without following any systematic model building discipline.. For example, the security requirements enforce two conditions that state that the messages will be transmitted in encrypted blocks and the server shall disconnect when a block is invalid in terms of content and structure. These requirements are modelled and verified in both empirical and view models using model-checking. As a general observation we concluded that the contracts in the view models produce better modular structure and de-coupled form in comparison with the empirical model. This has clear manifestation regarding the security requirements that are tighly coupled with the rest of the behaviour in the empirical model.

### 7.1 Test model construction effort

The advantages of the contract-based approach expose most clearly in model *development effort*. Although the empirical MBT approach proves to be useful (defects in design were found), incorporating large fragments of system behaviour in one monolithic test model increases the effort needed for model construction, comprehension and validation that may easily lead to the loss of model readability and each next development step requires exponentially more effort. In terms of *test model construction effort*, the CBD approach incorporates an intermediate structuring step to facilitate the creation of models by strictly following the view contracts that systematically guide selecting and representing model behaviours. In case of the *incremental multi-view test model construction*, the model development starts with a simple model that is enriched thereafter incrementally with design view specific information. Thus, the modelling is closely driven by the requirements via contracts unlike the relatively loose textual specification of the first approach.

In case of the *orthogonal multi-view test model construction* the view models can be developed similarly to incremental ones based on view contracts but there is a crucial difference, i.e., there is no direct need to superimpose the view models on top of some other already existing view model. Parallel model development has its productivity advantage. In addition, it enables more scalable correctness verification. As for integrating orthogonal view models there is freedom to choose by the test developer when the models will be conjoined in the process. However, the large size of conjoined models could complicate verification and test case definition. Therefore, one could prefer having more of feasible size view models instead of developing few of intractable size and keeping the conjoining operation to very last step.

### 7.2 Test model correctness verification effort

The *test model correctness verification effort* is determined by two factors: if only limited amount of information is added in the view model then this requires verifying only the effects introduced by the model increments. On the other hand, since newly introduced features are extending existing ones, the correctness queries of new features are strengthened each time, meaning satisfiability of later query implies also satisfiability of earlier ones, as exemplified in subsection 6.6. But rerunning all verification conditions on all increments may hit the complexity barrier before the complete model is composed from increments.

In contrast to limitations of incremental model verification, in case the model includes orthogonal views, like the security view in our case study, the correctness checking can be performed by applying model-checking on the conjunction of view models in a pairwise manner for better scalability (Aichernig et al., 2016).

Experience with climate control system case study confirmed that the CBD model verification effort was in average 93% and 98% more efficient (regarding model-checking time and memory consumed respectively) for the incremental timing view conjoined with the orthogonal security view compared to the empirical model development. In comparison the model-checking queries in both cases corresponded to the same system requirements. Moreover, the model verification effort was in average 99% more efficient for the behavioural view conjoined with the orthogonal security view compared to the empirical model verification time and memory consumed.

### 7.3 Test deployment effort

*Test deployment effort* considers primarily the development of test adapter to assist running the model againt the SUT. The simplicity of the model in the contract-based approach has benefits not only regarding test model development but also in reducing the size of test I/O-alphabet to be implemented in test case specific adapters. Our experience has shown that the multi-view contract-driven model development and related adapter building improves the overall testing efficiency by shortening the adapter development time almost exponentially in the size of symbolic test I/O-alphabet. Reduction of the input/output alphabets comes from the fact that view models that define the test cases have considerably smaller sets of input/output symbols than full monolithic test models. As for the testing tool set up adjustment to test cases, the tool DTRON (Anier et al., 2017) used in this approach requires only minor changes in time scaling parameter settings to adjust the test execution to physical latency conditions.

### 7.4 Test execution effort

In terms of the *test execution effort*, the traces generated by checking the Uppaal model properties show that the view model properties generate 23% shorter traces in average than the empirical model for the same set of coverage requirements. This has promising implication on the regression test efficiency and testing effort for large set of tests. Especially, the incremental building of test cases has shown added value for bug traceability because the incrementing resultant model incorporates also the tests of the earlier tested

views. Beside the validity checking of view implementation correctness, this allows detecting if any of earlier tested features gets corrupted due to newly introduced view features (feature interaction testing). Such feature interaction bugs are usually most difficult to find. Similar effect has been exposed in running orthogonal view models as test cases and their conjunction for detecting view interference effects. Again, these tests have shown considerable effect regarding test length compared to that of tests of the same bug detection power ran on monolithic model.

Considering the direct practical gain in our case study, both testing approaches show qualitatively equal bug detection capability. Both approaches detected that the climate control server fails to comply with requirements R1-R3 (stated in Section 2). The results show: (i) As opposed to R1, the server remains connected even if *client_hello* message is received later than $I_{TO}$ seconds after TCP connection is established. (ii) The server fails to disconnect after receiving a *client_hello* message with an incorrect value in one of its field in contrary to R2. (iii) In a scenario where the *client_hello* is sent in parts, the server disconnects the controller before receiving the complete message as opposed to R3. Though the error detection capabilities of applied techniques have shown to be equivalent on the case study, the view contract-based test cases address the violated requirements more clearly. This enlightens the back tracing to what requirements and how they have been violated based on the views, and thus, improving the *Root cause analysis effort*. Error debugging in the code becomes even more straightforward when the code is implemented by strictly following the same design view contracts like the test model.

## 8 Related work

The integration of assume/guarantee-based reasoning with automata-based formalisms has aimed at efficient verification through compositional reasoning. Compared to some influential works in the field (David, Larsen, Legay, Nyman and Wasowski, 2010; David, Larsen, Nyman, Legay and Wasowski, 2010; Larsen et al., 2006), we use an existing and widely accepted automata-based formalism for verification of real-time systems without conceptually extending it. Rather, we elaborate on how the contract meta-theory (Benveniste et al., 2015) corresponds to Uppaal TA via the mapping of system requirements to informal A/G contract patterns and then to TCTL properties that formalize these patterns as temporal saturated contracts. As a new contribution, we also elaborate on temporal multi-view contracts and show how our design multi-view approach covers the two main operators, composition and conjunction. Further, we adopt a more natural contract-based encoding of assumptions and guarantees, while the above works encode the assumptions and guarantees on the models directly without introducing view contracts as an intermediate model construction step.

The idea of A/G-based testing of software was proposed in (Blundell et al., 2005) where testing was performed on code generated from LTS models. The main difference compared to our work is that we use more expressive Uppaal TA formalism for representing SUT and verifying it against multi-view contracts. More recently, Boudhiba et al. applied MBT with contracts on program call level (Boudhiba et al., 2015). Instead, our approach can be applied at any system development phase provided the test inter-

face is well-defined and accessible to test adapters. A MBT approach similar to ours is presented in (Aichernig et al., 2016). The authors proposed the requirement interfaces formalism to represent different views of synchronous data-flow systems. Similar to our approach, they apply an incremental test generation procedure targeting a single system view at a time. Main differences compared to our approach are: 1) Direct encoding of contract assumptions and guarantees on the model, like in (David, Larsen, Legay, Nyman and Wasowski, 2010; David, Larsen, Nyman, Legay and Wasowski, 2010); 2) usage of different formalisms targeting different system domains while we apply only Uppaal TA relying on its relevant expressive power regarding CPS; and 3) the use of single modelling formalism instead of multiple domain specific languages allows us to limit to a single tool set, namely UPPAAL, that provides support to almost all required test development steps.

This article is an extended version of our paper from the proceedings of the 15th International Baltic Conference on Digital Business and Intelligent Systems (DB&IS) (Guin et al., 2022). In this extended version we provided a more thorough elaboration of the modelling process with explicit description of the multi-view model development scenarios, namely the *incremental* and *orthogonal* model construction operation applications. We then extended the case study by developing the test model of an orthogonal design view and integrating it with the previously developed test model with incremental model views. Finally, we updated accordingly the comparison of empirical and multi-view contract-based approach to developing test models.

## 9 Conclusion

The work compares two approaches used in system verification by MBT. In the empirical approach the model is created directly from unstructured requirements specification. The alternative method proposed in this paper exploits design view contracts as an intermediate step from requirements to models. The merits of the contract-based approach are evident as it makes the model creation and its verification process more structured and reduces the debugging effort due to the modular and incremental nature and provides explicit correctness conditions of view contracts that guide test model construction by intuitive and manageable with testing tools steps. The usability of proposed approach has been validated on an industrial climate control system testing case study where Uppaal tool family has been used for test model construction, model correctness verification, test deployment with adapters and test execution. The result of the case study shows the feasibility of this approach in practically all test development phases. One challenge however is the creation of contracts themselves from the weakly organized requirements specifications. As a future work a solution can be explored to reduce this effort by using structured language analysis for requirements extraction and specification patterns (Autili et al., 2015) for formulating the contract from requirements.

## Acknowledgement

This work has been supported by EXCITE (2014-2020.4.01.15-0018) grant.

## References


Aichernig, B. K., Hörmaier, K., Lorber, F. L., Nickovic, D., Tiran, S. (2016). Require, test, and trace it, *International journal on software tools for technology transfer* .

Anier, A., Vain, J., Tsiopoulos, L. (2017). DTRON: a tool for distributed model-based testing of time critical applications, *Proceedings of the Estonian Academy of Sciences* **66**, 75–88.

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A. (2015). Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar, *IEEE Transactions on Software Engineering* **41**(7), 620–638.

Baier, C., Katoen, J.-P. (2008). *Principles of Model Checking*, The MIT Press.

Behrmann, G., David, A., Larsen, K. G. (2004). A turorial on uppaal, *in* Bernardo, M., Corradini, F. (eds), *Proceedings of Formal Methods for the Design of Real-Time Systems, SFM-RT*, Vol. 3185 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 200–236.

Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K. G. (2015). Contracts for Systems Design: Theory, *Research Report RR-8759*, Inria Rennes Bretagne Atlantique ; INRIA.
https://hal.inria.fr/hal-01178467

Blundell, C., Giannakopoulou, D., Păsăreanu, C. (2005). Assume-guarantee testing, *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems*, SAVCBS '05, Association for Computing Machinery, New York, NY, USA, pp. 1–5.

Boudhiba, I., Gaston, C., Gall, P. L., Prévosto, V. (2015). Model-based testing from input output symbolic transition systems enriched by program calls and contracts, *in* El-Fakih, K., Barlas, G. (eds), *Proceedings of the 27th IFIP WG 6.1 International Conference on Testing Software and Systems - Volume 9447*, ICTSS 2015, Springer-Verlag, Berlin, Heidelberg, pp. 35–51.

David, A., Larsen, K. G., Legay, A., Nyman, U., Wasowski, A. (2010). ECDAR: An environment for compositional design and analysis of real time systems, *in* Bouajjani, A., Chin, W.-N. (eds), *Proceedings of Automated Technology for Verification and Analysis, ATVA*, Vol. 6252 of *LNCS*, Springer, Berlin, Heidelberg, pp. 365–370.

David, A., Larsen, K. G., Nyman, U., Legay, A., Wasowski, A. (2010). Timed i/o automata: a complete specification theory for real-time systems, *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, Association for Computing Machinery, New York, NY, USA, pp. 91–100.

Dias-Neto, A. C., Matalonga, S., Solari, M., Robiolo, G., Travassos, G. H. (2017). Toward the characterization of software testing practices in south america: looking at brazil and uruguay, *Software Quality Journal* **25**(4), 1145–1183.

Guin, J., Vain, J., Tsiopoulos, L., Valdek, G. (2022). Temporal multi-view contracts for efficient test models, *in* Ivanovic, M., Kirikova, M., Niedrite, L. (eds), *Digital Business and Intelligent Systems*, Springer International Publishing, Cham, pp. 136–151.

Larsen, K. G., Mikucionis, M., Nielsen, B. (2004). Online testing of real-time systems using uppaal, *in* Grabowski, J., Nielsen, B. (eds), *Formal Approaches to Software Testing, FATES*, Vol. 3395 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg.

Larsen, K. G., Nyman, U., Wasowski, A. (2006). An interface theory for input/output automata, *BRICS Report Series* **13**(11).
https://tidsskrift.dk/brics/article/view/21916

*PractiTest: The 2022 State of Testing Report* (n.d.). https://www.practitest.com/state-of-testing/.

Törngren, M., Sellgren, U. (2018). Complexity challenges in development of cyber-physical systems, *in* Lohstroh, M., Derler, P., Sirjani, M. (eds), *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, Springer International Publishing, Cham, pp. 478–503.

Utting, M., Pretschner, A., Legeard, B. (n.d.). A taxonomy of model-based testing approaches, *Software Testing, Verification and Reliability* **22**(5), 297–312.

Received November 6, 2022 , accepted November 29, 2022